SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 15(1), 59–64 (JANUARY 1985)

# An Interactive Assembly Level Debugging System

S. PANCHAPAKESAN, S. SUBRAMANIAN AND H. VENKATESWARAN
*National Aeronautical Laboratory, Bangalore-560017, India*

## SUMMARY

**An interactive assembly level debugging system has been developed to facilitate program development on an INTEL 8080A/8085 based microcomputer. It has features such as decoding machine level instructions into the assembly language, relocating programs in memory, changing instructions interactively at assembly level etc. This paper deals with the design of the assembly level debugging system and the various facilities and features it provides. The debugging system requires only 4·5K bytes of RAM besides the memory requirements of the application program that has to be debugged.**

KEY WORDS Interactive assembler Assembly debugging system Microcomputers

## INTRODUCTION

An interactive assembly level debugging system has been developed for an INTEL 8080A/8085 based microcomputer. The design of the debugging system and its features are presented. A program developed in an assembly language is first assembled and loaded into the microcomputer memory for subsequent execution. When the assembled program does not execute properly or as desired by the programmer, it becomes necessary to have a tool to facilitate debugging. Usually, the tools for debugging need knowledge of machine code, as all debugging monitors provided by manufacturers or developed individually work at machine code level with short mnemonic commands. These monitors decode the instructions/data into hex or octal code for display. So it is mandatory to know the hex or octal codes in order to carry out debugging on a mchine code program. Though these debug monitors provide very many features to facilitate debugging of programs, they on the other hand expect a good deal of machine code knowledge on the part of the user to interpret correctly the codes and data.

The interactive assembly level debugging system provides facilities for debugging programs at the assembly level itself. One such facility is to reconvert portions of the assembled program to the assembly language to know what exactly are the instructions in memory and to pin-point the bugs. The other facilities are modifying instructions/portions of a program, moving program units in the memory of the computer to test such units and to do patching of these units after the successful completion of tests, and to do relocation of programs without invoking re-assembly.

## THE DEBUGGING SYSTEM

The debugging system has been designed to convert machine code programs of INTEL 8080/8085 based microcomputers into assembly language instructions. The

0038–0644/85/010059–06$01.00
*Received 14 March 1980*
*Revised 13 September 1982*

assembly language mnemonics used for reconversion of machine code programs are as described in Reference 1. The characteristic byte table (CBT)[1] is extensively used to achieve conversion back to assembly language. The debugging system requires 4·5K bytes of RAM besides the memory requirements of the user programs.

After assembly and loading of the application program into the memory of the microcomputer, the debugging system can be invoked. A prompt character(?) appears on the TTY or a display terminal, prompting the user to enter the debug commands. The mnemonic commands provide many useful features such as copying, converting back to assembly mnemonics and relocating programs all at assembly level. Thus the mnemonic commands help development of several program units and integration of these into a single large program. The program units can be developed and/or tested using the debug facilities. Large and complex programs can be split into small program units and can be tried for various types of inputs and conditions. Also the grafting (patching) of working program units into the main-line program is easily done with the powerful commands of the debugging system.

## DESIGN OF THE INTERACTIVE ASSEMBLY LEVEL DEBUGGER

When an instruction in memory has to be converted back into the assembly language, the debug system looks at the opcode and identifies the type of instruction from its characteristic byte. The characteristic byte consists of information bits such as the length of the instruction, the number of opcodes involved and the position bits of the operands in the instruction word. The use and description of the characteristics byte can be found in Reference 1.

Once the type of instruction to be assembled into mnemonic form is identified, the debug system follows strictly the assembly language syntax rules which are embedded in it. Then the characters required for mnemonic opcodes, the operands, the directives, separators and other delimiters completing the syntax structure of the language are generated.

## SEARCH FOR THE OP-CODE

The opcode table, consisting of the mnemonics, machine codes and their corresponding characteristics bytes, is arranged in a specific order to facilitate scanning. A machine code to be assembled into mnemonic form is taken and compared with the entries in the opcode table, and if a match is found, the debug system uses the CBT information to assemble back the instruction in its mnemonic form.

This matching can be done in a single sweep for certain instructions such as RAL, RAR, PPP, PPS, BRA, etc. and the generation of the mnemonics can be carried out immediately. The instructions in this group contain only the operation code bits, and no operand bits are embedded in the instruction word. A list of such simple instructions that can be identified in a single sweep over the opcode table is provided in Appendix I.

There are other instructions which contain operand information, such as the register number or condition flags. These instructions will not obtain a match as the opcode table entries do not contain these information bits.

Hence it becomes necessary to assume certain bit patterns and remove the appropriate information bits in the instruction word before scanning the entries in the opcode table for a match.



It is found that there are only six types of masks that are required to filter out the information bits and produce the 'pure' opcode' These six types of masking are unique to the instruction set of INTEL 8080A/8085. The mask bits for these instructions are provided in Appendix II.

Once the opcode is identified, its characteristics byte information is available, the information bits filtered earlier with masks are used to generate the appropriate operands and the syntax structure of the assembly language. If the instruction happens to be a multibyte instruction the subsequent byte/bytes are mere data and they are converted into Hex code. Then their assembly instructions and data are generated in that order. Since the CBT gives all information regarding the instruction, such as the length, the number of operands and the position bits, the characteristic bytes for such instructions are used by the debug system to generate the appropriate mnemonics to accomplish the assembly language equivalents.

## RELOCATION OF PROGRAMS

When programs are copied in memory they have to be relocated. Apart from relocating the jump and call addresses, it may be necessary to relocate some addresses used in instructions such as LRP H, A..... LSP SP, A....., also. These instructions may have been used to define data areas for the program.

Hence two types of commands are provided for relocation. The CPYRLC command followed by a subsequent command PART copies and relocates program units, effecting relocation only for branch instructions such as BRA, BCT, BCF and for call instructions such as CAL, CCT, and CCF. In the case when data-addresses have to be replaced, the command CPYRLC followed by FULL must be used.

However, the FULL command should be exercised with caution, as it would inadvertently change the codes in the Load Register Pair instructions which are intended to represent immediate data and not pointers to data area.

## DEBUG SYSTEM COMMANDS

The commands use six character names to provide the following functions:

1. copying program units in memory and relocating them, partly or fully (CPYRLC PART/FULL)

2. decoding and converting the machine instructions at the specified address in memory to assembly language (MNEASM MNENXT)

3. assembling an assembly language program from a specified address in memory (ASMBLE)

4. producing listings of programs in the assembly language for documentation (GODOCM)

5. displaying the contents of all registers (DSPALL)

6. executing instructions in the memory (EXECUT).

The exact commands and examples of these functions are listed in Appendix III.

## CONCLUSION

The debug system is useful for developing/debugging programs at the assembly language level. The debug system has been working quite satisfactorily for the past two years on an INTEL 8080A based system located in our Laboratory.

ACKNOWLEDGEMENTS

The authors wish to thank the Director, NAL for his kind permission to publish this paper. The authors wish to thank the referee for his valuable comments and criticism in the light of which this paper has been revised.

## APPENDIX I

Opcode mnemonics that can match directly on the first sweep with the entries in the opcode table are:

| | | | |
|---|---|---|---|
| RAL | CMA | NOP | CAL |
| RAR | CMC | ADI | LAD |
| RLC | SEC | ACI | SAD |
| RRC | JHL | SUI | LHM |
| RET | PPS | SBI | SHM |
| HLT | PPP | NDI | INP |
| EXH | EST | XRI | OUP |
| INM | MHS | IRI | |
| DEM | ENI | CPI | |
| DAA | DBI | BRA | |

## APPENDIX II

| Type number | Mask (octal) | Opcodes that get a match |
|---|---|---|
| 1 | 370 | ADD, ADC, SUB, SBB, AND, EXR, INR, CMP |
| 2 | 317 | BCT, CCT, RCT |
| 3 | 307 | RST, PRP, PPR, BCF, CCF, RCF |
| 4 | 300 | TRN |
| 5 | 017 | ORP, ARP, LAI |
| 6 | 007 | LDA, INC, DEC, LRP, IRP, SAI |

## APPENDIX III

**Second level commands in monitor**

1. CPYRLC XXXX YYYY
   FULL

Copy the program from location XXXX (hex) to the location YYYY (hex) and relocate fully all the addresses of jump and call instructions as well as LRP instructions.



*Example*

A program from location 0200 (hex) reads:

```
0200  LRP  H,H0600
0203  CAL  H0252
0206  CAL  H0280
0209  BCT  Z,H020D
020C  HLT
020D  CAL  H0240
0210  BRA  H0400
```

Now the command is executed:

```
? CPYRLC  0200  0300
  FULL
?
```

The result will be

```
0300  LRP  H,H0700
0303  CAL  H,0352
0306  CAL  H,0380
0309  BCT  Z,H030D
030C  HLT
030D  CAL  H0340
0310  BRA  H0500
```

2. CPYRLC XXXX YYYY
   PART

This copies the program from location XXXX (hex) to location YYYY (hex) but relocates only the jump and call addresses.

*Example*

In the previous example, if the PART command is used, the result will be the same *except* for the first line which will read as LRP H,H0600. The rest of the program will be as in the previous result.

3. MNEASM XXXX

Convert the instruction in the memory location XXXX (hex) into assembly language.

4. MNENXT XXXX

Convert the instruction in the next location, into assembly language.

*Example*

Let the contents of the memory locations 0200 to 0206 be as follows:

(0200) = 21
(0201) = 00
(0202) = 04
(0203) = CD
(0204) = 52
(0205) = 02
(0206) = 76

When the command is executed as

```
? MNEASM 0200
```

the result is

```
0200  LRP  H,0400
? MNENXT
0203  CAL  H0252
? MNENXT
0206  HLT
?
```

5. ASMBLE XXXX

Assemble a program to be input from memory location XXXX (hex) till a directive FIN is encountered.

*Example*

```
? ASMBLE 0200
0200  LRP  H,A1024
0203  CAL  A6594
0206  HLT
0207  FIN
```

Here the addresses on the left are printed by the assembler part of the monitor. Instructions are given by the user through keyboard, which are assembled till the FIN directive.

6. GODOCM XXXX YYYY

Print out the program residing in the memory from location XXXX to location YYYY in assembly language.

*Example*

```
? GODOCM 0200 0206
0200  010004     LRP  B,H0400
0203  CD 15 02   CAL   H0215
0206  76         HLT
```

The output consists of the memory address, the machine code and the assembly language statement. This will be useful for documenting. Note that labels and subroutine names cannot appear in this listing but only absolute addresses.

7. DSPALL

Display the contents of all the register.

*Example*

```
? DSPALL
A = 15   B = 20  C = 25  D = 30
E = 35   F = 42  H = 40  L = 60
?
```

8. EXECUT

Execute the program from its starting location defined in the start of the debugging/developing session.

## REFERENCE

1. S. Panchapakesan, H. Venkateswaran and S. Subramanian, 'Assemblers for micro-computers', *Software—Practice and Experience*, **9**, 843–852 (1979).

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 15(1), 65–86 (JANUARY 1985)

# An IKBS Implementation

PHILIP LEITH

*Faculty of Mathematics, Open University, Milton Keynes MK7 6AA, U.K.*

**SUMMARY**

**This paper details the practical approaches and solutions taken in the implementation of an INTERLISP programmed knowledge based program. Although some of the solutions used are particular to the specific problem domain of this program, the general pragmatic methods are of wider interest. The program excerpts are given in Pseudo-LISP code.**

KEY WORDS IKBS INTERLISP

## INTRODUCTION

Much of the currently available literature on IKBS techniques describes systems in relatively high level terms. This situation is unfortunate in many ways, but most particularly because there is a very necessary requirement for basic introductory writings on the tools and techniques rather than the goals and general methodologies towards which IKBS systems designers are working. The techniques and tools are, in fact, just those which are used in other areas of computing—such a fact is often obscured by high level system descriptions. In this paper I describe the actual implementation of a system which has been discussed previously at a higher level. Although this system is not representative of all currently available systems, it does accord with the generic 'expert system' type.

There are basically three problems in the design of an IKBS system. The first, and most difficult, is to choose the data structures which will represent the area of expertise handled by the system. As with all computer programs it is necessary to choose one of several possible structures. Each of the possible data structures might be advantageous in solving one particular problem, but not another. So it is with IKBS programs—one might use production rules, another frames and a third, associative networks. Just as likely as a system which uses only one data structure to handle its expertise is a system which uses more than one (EMYCIN uses a tree structure—called a context tree—as its prime data structure; it is built and transformed by production rules held in a different structure).

The second area is that of interpretation—how can the information which is held in these data structures be extracted from it.

The third area is that of filling up the data structures with the information which, when interpreted, is to become the system's expertise.

The ELI system was designed to handle one type of information—legislation covering British Welfare Rights. The legislation is transformed by an expert within the field into rules (sometimes called production rules or productions). The general approach is outlined in Reference 2. In this text, I deal with the handling of rules

0038-0644/85/010065-22$02.20

*Received 7 December 1983*
*Revised 17 May 1984*